CLASS

the Cosmological Linear Anisotropy Solving System[1]

Julien Lesgourgues
TTK, RWTH Aachen University

CCA, New York, 15-16.07.2019

[1] code developed by Julien Lesgourgues & Thomas Tram plus many others...

class in New York

• Basics:	Coding spirit and general rules
• Theory:	Physical assumptions in the code
• Usage I:	From interactive runs to python scripts / jupyter notebooks
• Usage II:	Exploring the code possibilities through more advanced notebooks
• Coding I:	Essential rules and conventions specific to the code
• Coding II:	How to implement new physics and new ingredients
• News:	Expected content of next releases and more long-term projects
• Exercise I:	Extracting and plotting various quantities
• Exercise II:	Modifying the code

Context

class is the 5th public Einstein-Boltzmann solver covering all basic cosmology:

1. COSMICS package in f77 (Bertschinger 1995)
   Basic equations, brute-force C_l^{TT}
2. CMBFAST in f77 (Seljak & Zaldarriaga 1996)
   Line-of-sight, $C_l^{EE,TE,BB}$, open universe, CMB lensing
3. CAMB in f90/2000 (Lewis & Challinor 1999)
   closed universe, better lensing, new algorithms, new approximations, new species, new observables... (http://camb.info)
4. CMBEASY in C++ (Doran 2003)
5. class in C (Lesgourgues & Tram 2011)
   simpler polarisation equations, new algorithms, new approximations, new species, new observables... (http://class-code.net)

... and there might still be 1 or 2 more! But only CAMB and class are currently developed and kept to high precision level.

Context

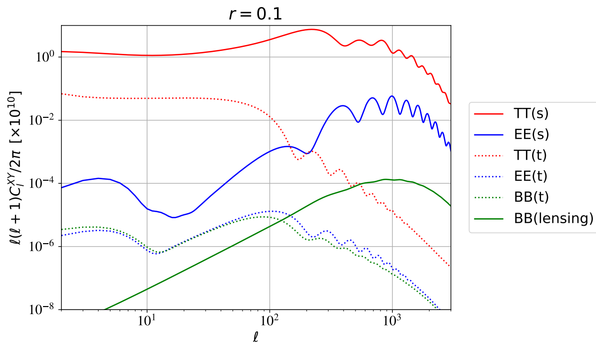
Project started on request of Planck science team, in order to have a tool independent from CAMB, and check for possible Boltzmann-code-induced bias in parameter extraction. The `class`-CAMB comparison has triggered progress in the accuracy of both codes. Agreement established at 10^{-4} (0.01%) level for CMB observables, using highest-precision settings in both codes. But the `class` projected expanded and went much further than the initial Planck purposes.

`class` aims at being:

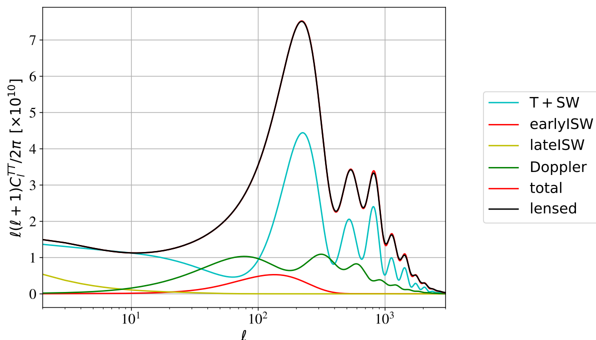
- general (more models, more output/observables)
- modern (structured, modular, flexible, wrap-able: wrapper for python, C++, automatic precision test code)
- user friendly (documented, structured, easy to understand) and hence easier to modify (coding additional models/observables)
- accurate and fast (currently comparable to CAMB; in principle, clear structure offers potential for further optimisation/parallelisation/vectorisation/etc.)

With `class` you can get:

The CMB anisotropy spectra:

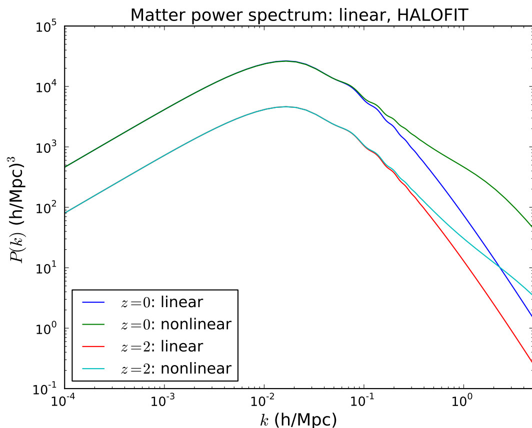


The CMB temperature spectrum decomposition:



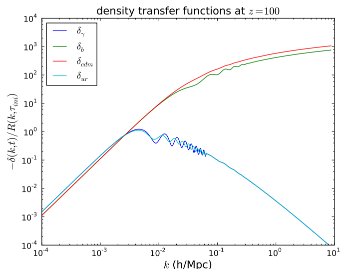
With `class` you can get:

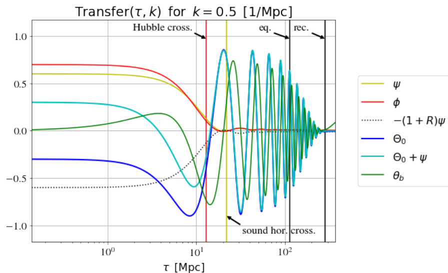
The matter power spectrum:



With `class` you can get:

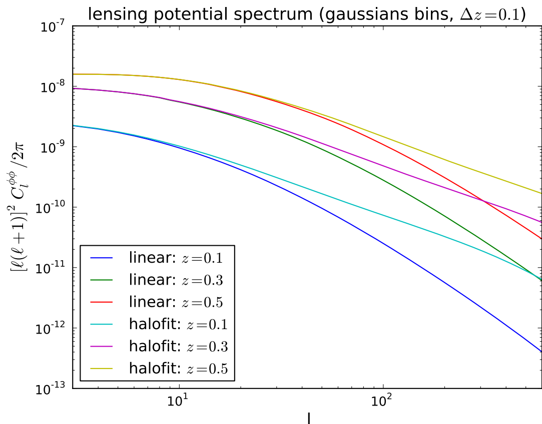
The transfer functions at a given time/redshift (e.g. initial conditions for N-body):





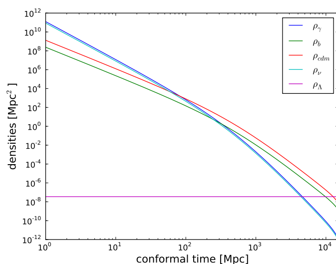
With `class` you can get:

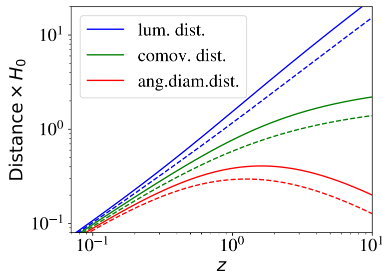
The matter density (number count) C_l's, or the lensing C_l's (with arbitrary selection/window functions):



With class you can get:

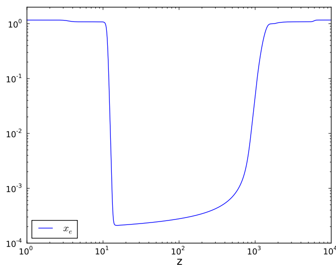
The background evolution in a given cosmological model:

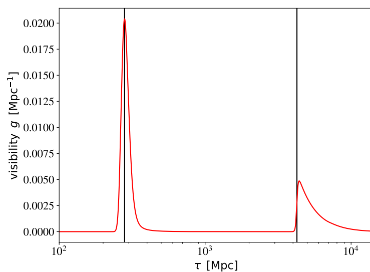




With `class` you can get:

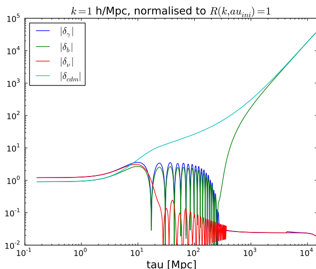
The thermal history in a given cosmological model:

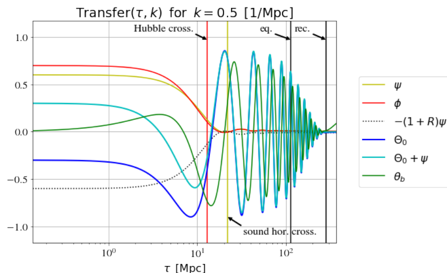




With `class` you can get:

The time evolution of perturbations for individual Fourier modes:





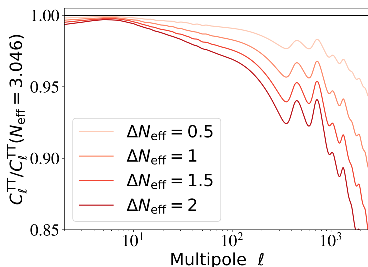
With class you can get:

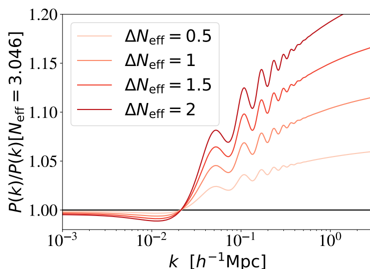
… and several other quantities, for instance:

- distance–redshift relations, sound horizon, characteristic redshifts;
- primordial spectrum for given inflationary potential;
- decomposition of CMB C_l's in intrinsic, Sachs-Wolfe, Doppler, ISW, etc.;
- decomposition of galaxy number count C_l's in density, RSD, lensing, etc.;
- …

With class you can get:

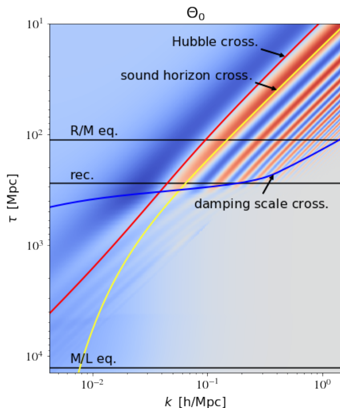
... if you use class as a python module you can extract all kind of output or intermediate quantities, manipulate them in various ways, and make all kinds of computations or nice plots:

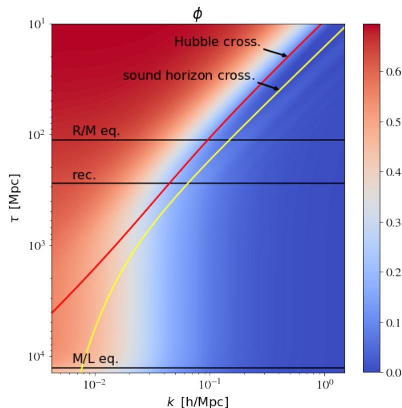




With `class` you can get:

... if you use class as a python module you can extract all kind of output or intermediate quantities, manipulate them in various ways, and make all kinds of computations or nice plots:





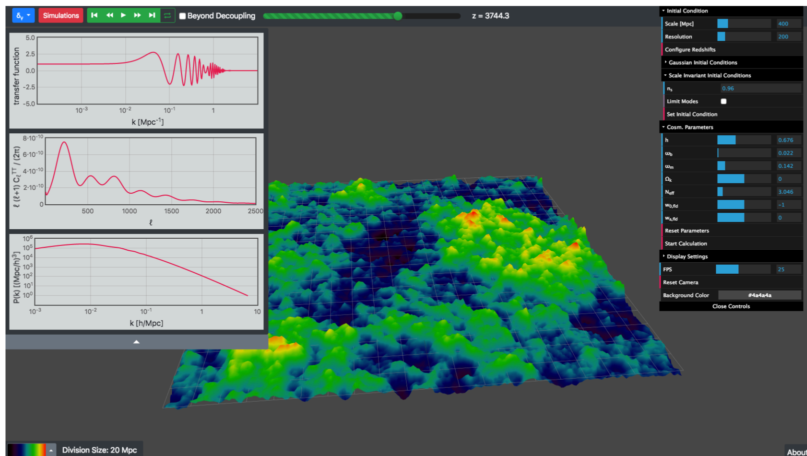
With `class` you can get:

... all this for a wide range of cosmological models: all those implemented in the public CAMB code, plus several other ingredients, especially in the sectors of:

- primordial perturbations (internal inflationary perturbation module with given $V(\phi)$, takes arbitrary BSI spectra, correlated isocurvature modes),
- neutrinos (chemical potentials, arbitrary phase-space distributions, flavor mixing...),
- Dark Matter (warm, annihilating, *decaying, interacting...*),
- Dark Energy (fluid with flexible $w(a)$ + sound speed, quintessence with given $V(\phi)$)
- also Modified Gravity if you try the recently released HiCLASS branch (Bellini, Sawicki, Zumalacarregui, `http://www.hiclass-code.net`)
- multi-gauge (synchronous, newtonian...)
- extension to second-order perturbation theory: SONG (Fidler, Pettinari, Tram, `https://github.com/coccoinomane/song`)
- interfacing with particle physics modules and codes for exotic energy injection available in ExoCLASS branch of `http://github.com/lesgourg/class_public.git` (Stöcker, Poulin)

With class you can get:

... and movies of CMB perturbations in 2D slices of early universe with our Real space graphical interface (part of public package since v2.7.0); here is a snapshot:



class coding spirit

Equations follow literally notations of most famous papers

(in particular Ma & Bertschinger 1996, `astro-ph/9506072`).
Multi-gauge code: everything coded in newtonian and synchronous gauge, structure ready for more gauges.

`class` coding spirit

Equations follow literally notations of most famous papers

(in particular Ma & Bertschinger 1996, `astro-ph/9506072`).
Multi-gauge code: everything coded in newtonian and synchronous gauge, structure ready for more gauges.

Input parameters interpreted and processed into final form needed by the modules

Some basic logic has been incorporated in the code. Easy to elaborate further.
Examples:
- expects only one out of $\{H_0, h, 100 \times \theta_s\}$, otherwise complains;
- missing ones inferred from given one
- same with $\{T_{\rm cmb}, \Omega_\gamma, \omega_\gamma\}$, $\{\Omega_{\rm ncdm}, \omega_{\rm ncdm}, m_\nu\}$, $\{\Omega_{\rm ur}, \omega_{\rm ur}, N_{\rm ur}\}$,...

class coding spirit

Equations follow literally notations of most famous papers

(in particular Ma & Bertschinger 1996, astro-ph/9506072).
Multi-gauge code: everything coded in newtonian and synchronous gauge, structure ready for more gauges.

Input parameters interpreted and processed into final form needed by the modules

Some basic logic has been incorporated in the code. Easy to elaborate further.
Examples:
- expects only one out of $\{H_0,\ h,\ 100\times\theta_s\}$, otherwise complains;
- missing ones inferred from given one
- same with $\{T_{\rm cmb},\ \Omega_\gamma,\ \omega_\gamma\}$, $\{\Omega_{\rm ncdm},\ \omega_{\rm ncdm},\ m_\nu\}$, $\{\Omega_{\rm ur},\ \omega_{\rm ur},\ N_{\rm ur}\}$,...

Homogeneous units

Inside all modules except thermodynamics: everything in Mpc^n.
Examples:
- conformal time τ in Mpc, $H=\frac{a'}{a^2}$ in Mpc^{-1}
- $\rho_{\rm class}\equiv\frac{8\pi G}{3}\rho_{\rm physical}$ in Mpc^{-2}, such that $H^2=\sum_i\rho_i$
- k in Mpc^{-1}, $P(k)$ in Mpc^3

`class` coding spirit

Accessible and self-contained

Plain C (for performance and readability) but mimicking features of C++ (see later).
No external libraries for a quick installation (but parallelisation requires OpenMP).
Lots of comments in the code.
Automatic `doxygen` documentation (Credits Deanna C. Hooper)

class coding spirit

Accessible and self-contained

Plain C (for performance and readability) but mimicking features of C++ (see later).
No external libraries for a quick installation (but parallelisation requires OpenMP).
Lots of comments in the code.
Automatic `doxygen` documentation (Credits Deanna C. Hooper)

Structured and flexible

Sequence of ten modules with distinct physical tasks, no duplicate equations.

class coding spirit

Plethoric accumulation of extended models/observables/features without making the code slower or less readable

Relies on homogeneous style and strict rules (e.g. anything related to given feature is inside an: `if (has_feature == _TRUE_){...}`)

`class` coding spirit

Plethoric accumulation of extended models/observables/features without making the code slower or less readable

Relies on homogeneous style and strict rules (e.g. anything related to given feature is inside an: `if (has_feature == _TRUE_){...}`)

No hard-coding

- All indices allocated dynamically (according to strict and homogeneous rules for more readability): see dedicated section in "Coding I" lecture
- All arrays allocated dynamically
- Essentially no number found in the codes except factors in physical equations
- No hard-coded precision parameters, all precision-related numbers/flags gathered in single structure `precision`
- Not a single global variable: all variables passed as arguments of functions (for readability and parallelisation)
- Sampling steps inferred dynamically by the code for each model
- Time for switching approximations on/off inferred dynamically by the code for each model

class coding spirit

Rigorous error management

In principle, no segmentation faults when executing public class.
When class fails, it returns an error message with a tree-like information (like e.g. python)
We'll see how this works in "Coding I" ...

class coding spirit

Rigorous error management

In principle, no segmentation faults when executing public `class`.
When `class` fails, it returns an error message with a tree-like information (like e.g. python)
We'll see how this works in "Coding I" ...

Version history

All previous versions can be downloaded and compared on GitHub, changes documented in `class-code.net`
Always aim at developing without breaking compatibility with older versions.
Own changes can often be merged in newer version with `git merge`.

Installation

Installation should be straightforward on Linux, and slightly tricky but still easy on Mac. We suggest to not even try on Windows.
We really recommend cloning the code from GitHub. The old-fashioned way, i.e. downloading a .tar.gz, also works.
In the ideal case you would just need to type in your terminal

```
> git clone http://github.com/lesgourg/class_public.git
    class
> cd class/
> make clean;make -j
```

and you would be done. To check whether the C code is correctly installed, you can type

```
> ./class explanatory.ini
```

which should run the code and write some output on the terminal. To check whether the python wrapper installation also worked, try

```
> python
>>> from classy import Class
>>>
```

and just check that python does not complain. If any of these steps does not work, please look at the detailed installation instructions at
https://github.com/lesgourg/class_public/wiki/Installation